

The background of the slide is a photograph of a dried, pressed leaf and a thin branch against a light-colored, textured paper. The leaf is positioned on the left side, with its stem extending downwards. Another smaller leaf is visible on the right side, partially obscured by the text.

Wrappers, Aspects, Quantification and Events

Robert E. Filman

Research Institute for Advanced
Computer Science

NASA Ames Research Center

rfilman@mail.arc.nasa.gov



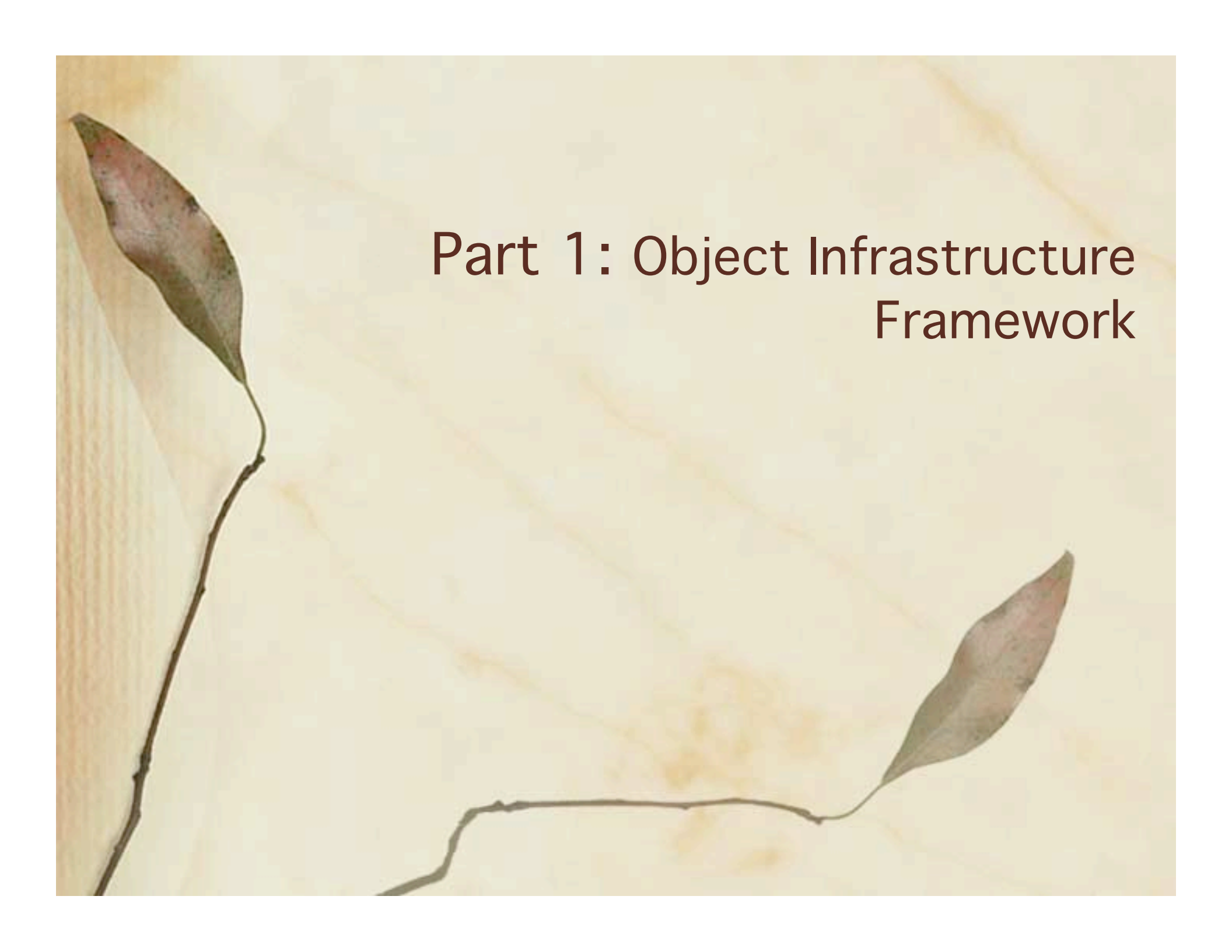
Talk Overview

- Object Infrastructure Framework (OIF)
 - A system developed to simplify building distributed applications by allowing independent implementation of multiple concerns
- Essence and State of AOP
- Trinity
 - Quantification over Events
 - Current work on a generalized AOP technology



Co-conspirators

- Stu Barrett, Diana Lee, Ted Linden
- Dan Friedman
- Klaus Havelund, Dave Herman, Jeff Palm
- Tzilla Elrad, Siobhán Clarke, Mehmet Aksit

The background of the slide is a light beige, textured surface, possibly paper or fabric, with subtle variations in tone and some faint, darker spots. On the left side, there is a vertical strip of light-colored wood grain. Two dried, pressed leaves are visible: one in the upper left corner, partially overlapping the wood, and another in the lower right quadrant. Both leaves are dark brown and have a slightly curled, elongated shape. The title text is positioned in the upper right area of the slide.

Part 1: Object Infrastructure Framework

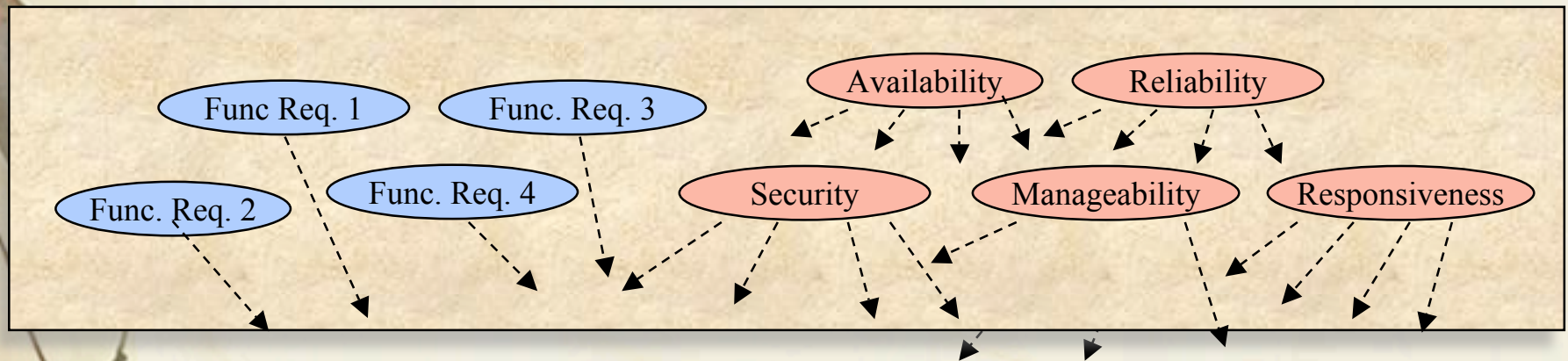


Object Infrastructure Framework (OIF)

- Microelectronics and Computer Technology Corporation (MCC) project to simplify building distributed systems (1997-98)
- Developing distributed applications is difficult
 - Hard to achieve systems with systematic properties, (ilities) e.g.:
 - Reliability
 - Security
 - Manageability
 - Quality of Service
 - Scalability
 - Distribution is complex
 - Concurrency is complicated
 - Distributed algorithms are difficult to implement
 - Every policy must be realized in every component
 - Existing frameworks are difficult to use

Functional and Non-Functional Requirements

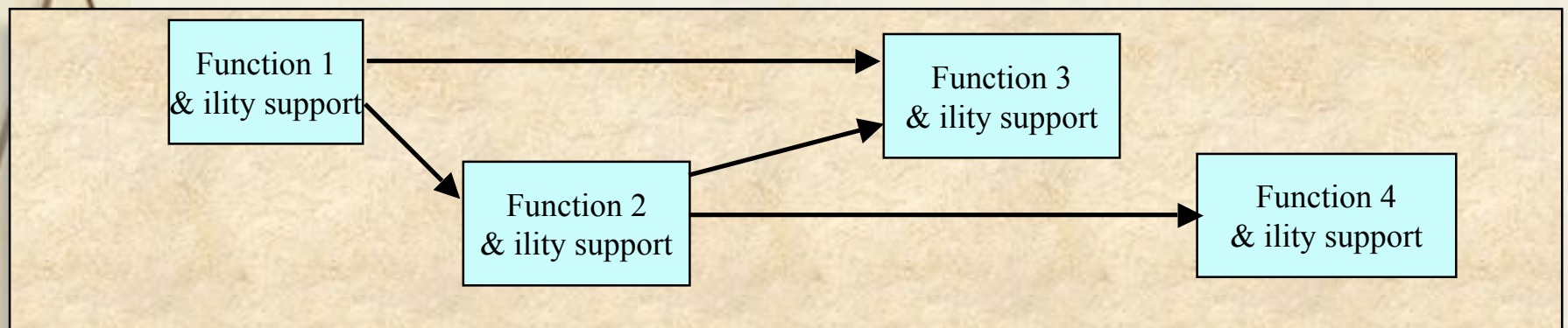
Functional:




**Functional requirements
map to specific components**

Non-functional (ility):

**ility requirements map
almost everywhere**





The Problem: Managing the Service Space

- Programmers when writing code have to keep cognoscente of when to invoke which ilities
- Compatible support for ility requirements (security, consistency, responsiveness, etc.) is a key component integration problem
- Ilties impose difficult upgrade requirements on component subsystems
 - Algorithms that support ilties are usually intertwined with the subsystem functional logic.
 - Separately developed subsystems may have chosen different algorithms (for encryption, transactions, etc.)
 - Close examination of the system required when evolving ilties

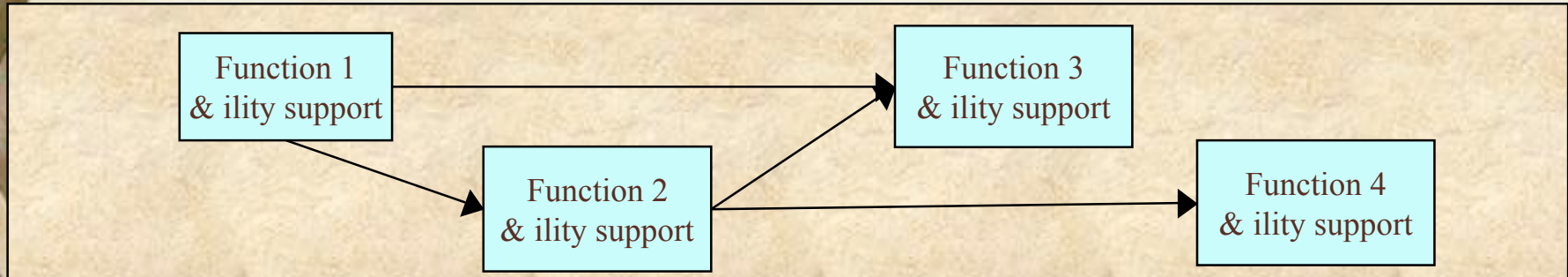
Research Hypothesis

- Ilities can be achieved by inserting services into the communication path between functional components
 - On both sides of the communication divide
- Frameworks that automate service insertion can systematically achieve non-functional requirements
 - Object Infrastructure Framework (OIF)

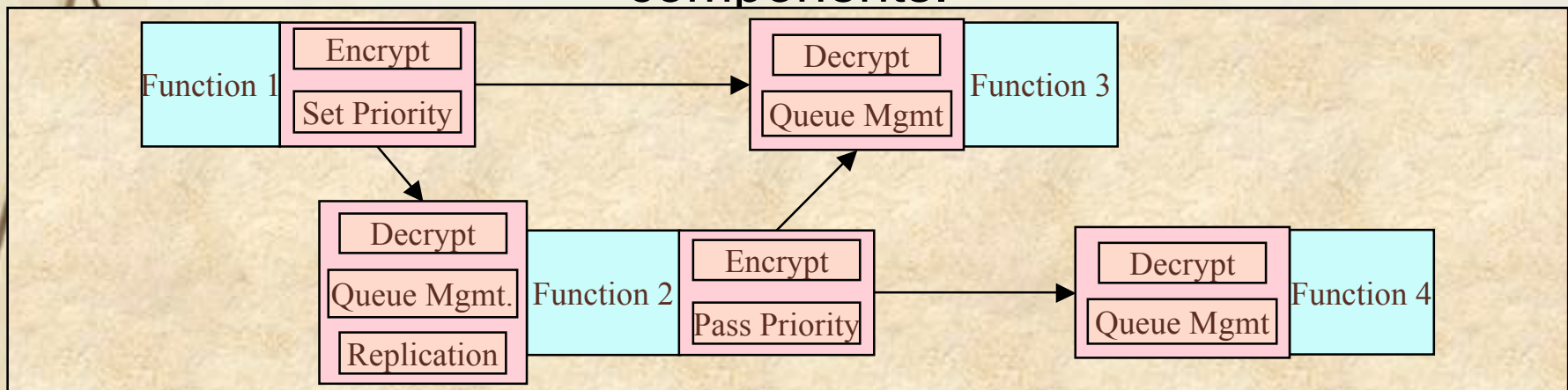


Architecture with Services in Component Communications

Traditional designs mix ility support within functional components



OIF separates service functionality from functional logic by inserting the services on the communications paths among components.





Distributed Object Technology

- Several approaches:
 - Socket based
 - Message based.
 - Remote Procedure Call
 - Web services
 - Object based
- Distributed Object Technology allows for OO applications to be implemented using *some* objects that do not reside in the same address space (virtual machine).
 - Key semantic of DOT is providing *location transparency*
- CORBA: Objects provide services described in an Interface Definition Language (IDL)
 - CORBA allows object-oriented applications to be written in multiple languages.
 - Java RMI is monolingual CORBA

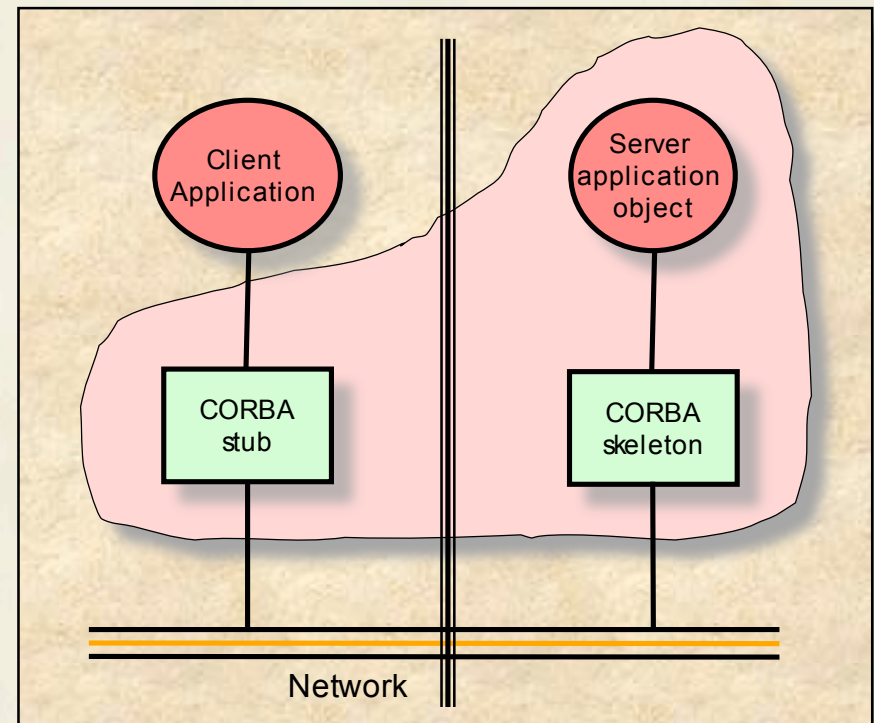
Using Stubs and Skeletons as Proxies

A client application makes a method invocation on the stub.

The Stub “implements” the IDL defined interface by being a *proxy* for the actual implementation object.

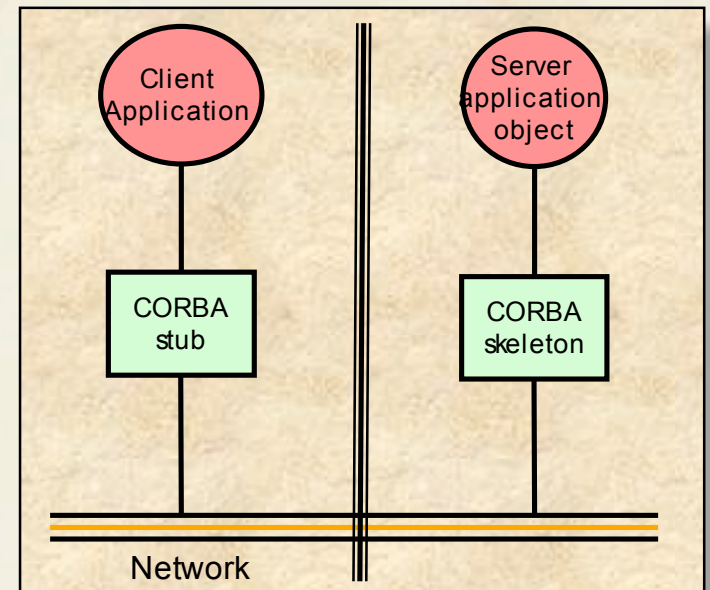
Similarly, the Skeleton acts as a proxy for the client application (from the implementation’s perspective).

The implementation object needs to have the actual code for each method defined the IDL interface.



How Proxies Work

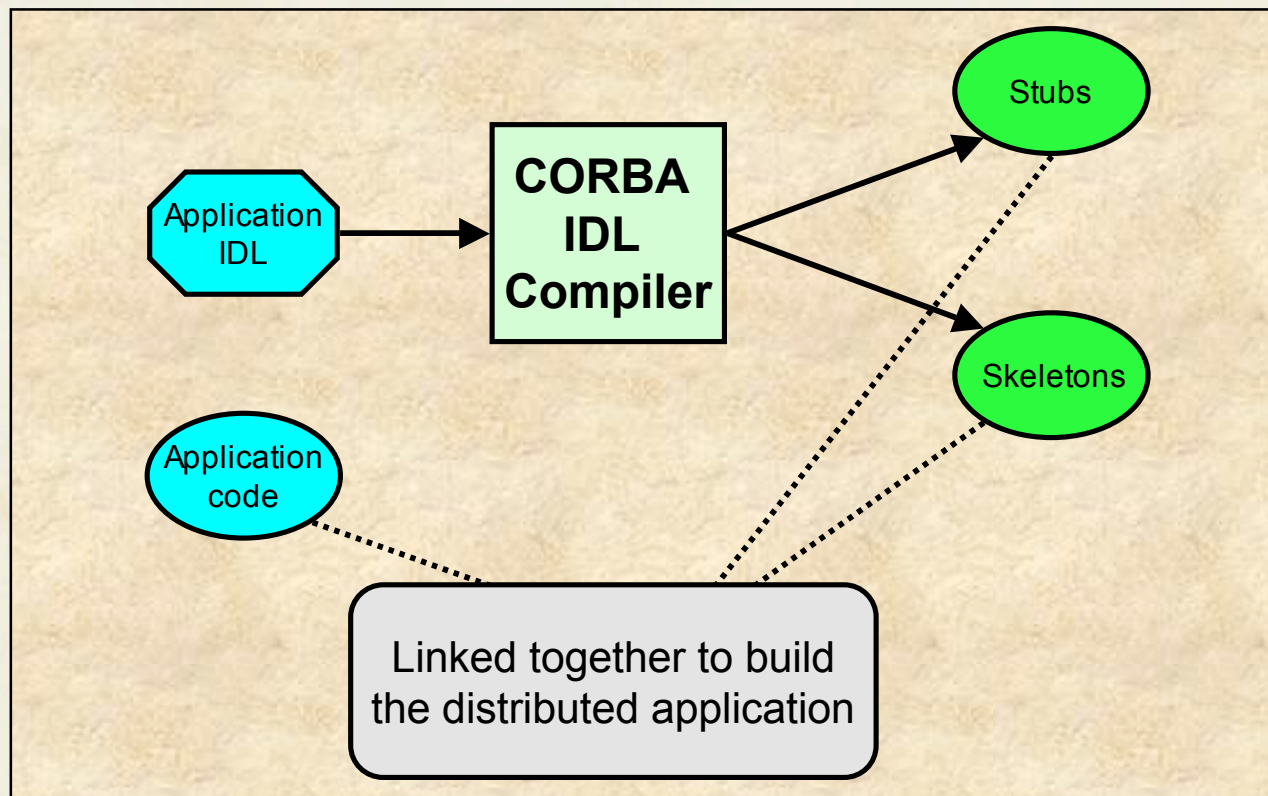
The stub “implements” the IDL defined interface. The operation’s arguments are put into a request object, marshaled and passed over the network to the server machine.



Marshaling handles all of the issues relating to transmission and translation of the various data types.

The ORB on the server machine demarshals the client’s request, and invokes the skeleton. The skeleton calls the appropriate implementation object method. The process is applied in reverse to the return value.

Compiling Stubs and Skeletons



The IDL compiler produces both client-side stubs and server side skeletons.



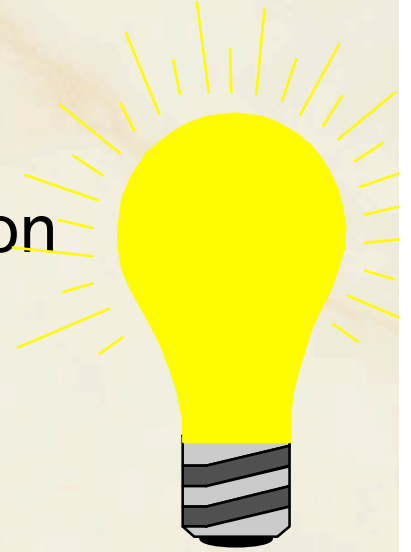
Distributed Object Proxies

- What they do:
 - Provide object location transparency
 - Hide details of communication protocols
- What they do *not* do:
 - Handle partial failures (reliability).
 - Security related issues.
 - Quality of service issues.
 - ...

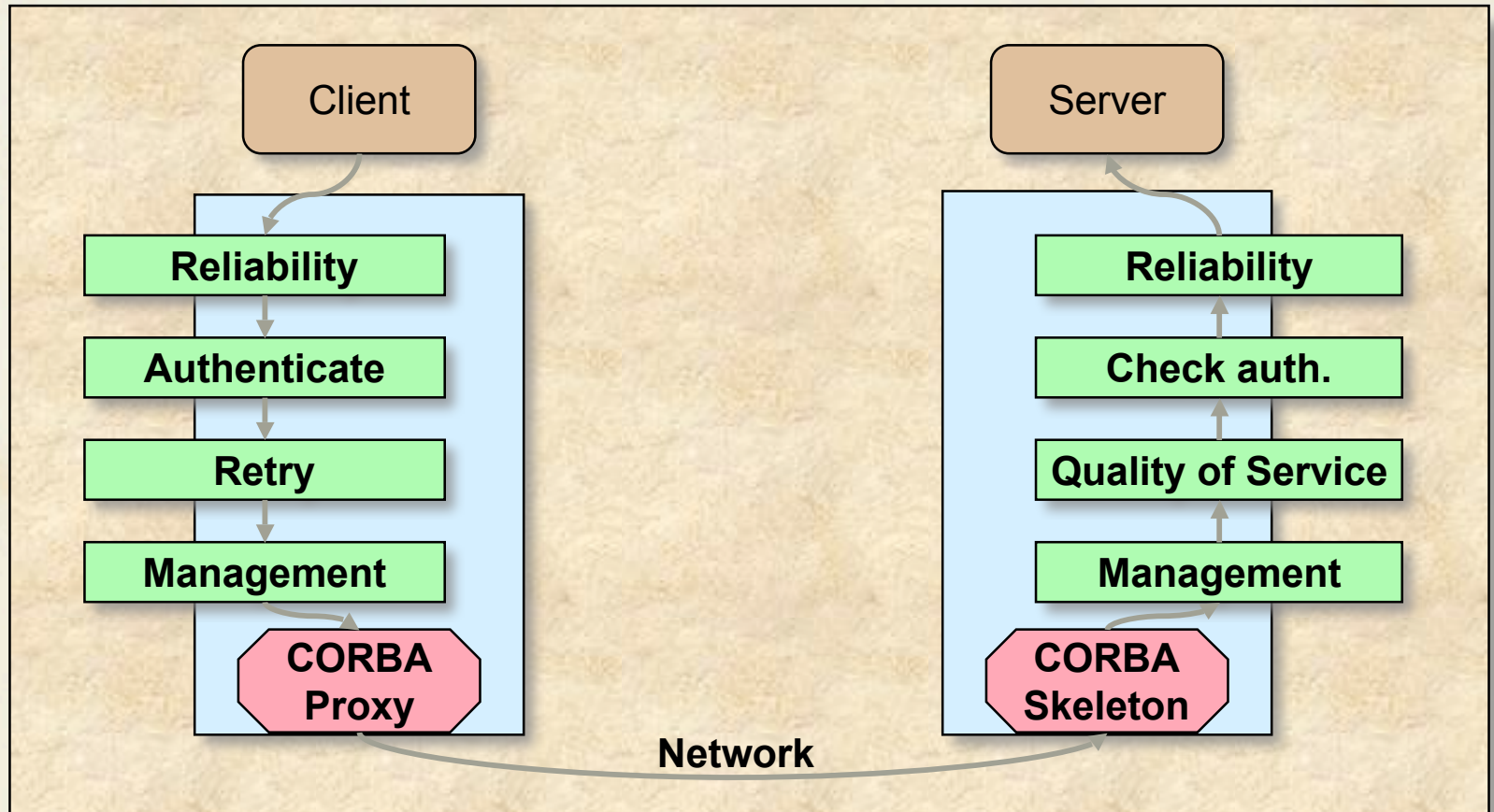
OIF Challenge: Can we leverage the “proxy” design to address these missing characteristics?

Key OIF Ideas

- *Injecting behavior* on the communication paths between components
 - Injectors are discrete, uniform objects
 - Injectors are by object/method
 - Injectors are dynamically configurable
- *Annotated communications* allow injected services to pass parameters to service peers (e.g., message priority, user-id, tracing status)
- *Thread contexts* preserve annotations through calls
- *Pragma*: High-level specification language for describing desired injections



Configurable Proxies



- ✧ OIF's injectors can operate in pairs (e.g., encrypt/decrypt; request authentication/authenticate) or singly (e.g., retry on failure; log results)
- ✧ Configuration is by proxy/method instance
- ✧ Configuration is dynamic



Injector Features

- Ability to access/mutate method arguments and return value.
- Ability to pass meta-data (property sets) between themselves to coordinate their behavior.
- Access to CORBA's DII and DSI services
 - Access/modify function arguments, return value
 - Change "target" of request (load balancing, reliability)
- Fully capable code module.
 - Can be multi-threaded.
 - Can access other objects/services
 - Throw/catch exceptions



Injector-enabled services

- Caching of static object attributes reduces repetitious remote requests and enables “delayed call by value.”
- Serialization of arriving requests reduces cognitive load on application developer.
- Reified requests allow reasoning about priorities
- Targets allow application components to invoke logical destinations rather than a specific object (so injected services can do load-balancing, replication, transaction processing, redirection, etc.)
- Futures enable asynchronous interaction between application components while writing synchronous code.

Reality

Ilities must be grounded in the reality of invoking actual services

Saying you want security doesn't produce security.

- Rather, you have to decide that you've got security if you
 - Encrypt all communications using { 64|128|3 } bit { DES | RSA | ROT-13 }
 - Check the user's { password | fingerprints | DNA } for { every | occasional } access to { all | only sensitive } methods
 - Recognize intrusions that { come from strange sites | try a series of passwords | ask too many questions }
 - Keep track of privileges by { proximity | job function | dynamic agreements }
- Need to have (implementations) of the algorithms
- Need to know where which algorithms are to be applied in which circumstances



Pragma: OIF's Quantification Language

○ Problem:

□ Locally:

- Arranging for the appropriate injectors to be on the appropriate methods in the right order for each proxy
- Precluding incompatible injectors

□ Globally:

- Achieving ilities

○ Solution:

□ Pragma: A high-level, declarative specification language for defining

- Ilities
- Ways to achieve ilities (i.e. which injectors + parameters to run to get that ility)
- The mapping for each ility to the methods of the application objects

□ Pragma compiler:

- Takes declarative specification and compiles Java injector initializations

This is policy vendoom.

These are namespace imports.

Vendoom uses five ilities. For each ility, class and method, there (may) be more than one way to achieve that ility.

Var declares annotations, their types, default values and when they're copied to thread contexts.

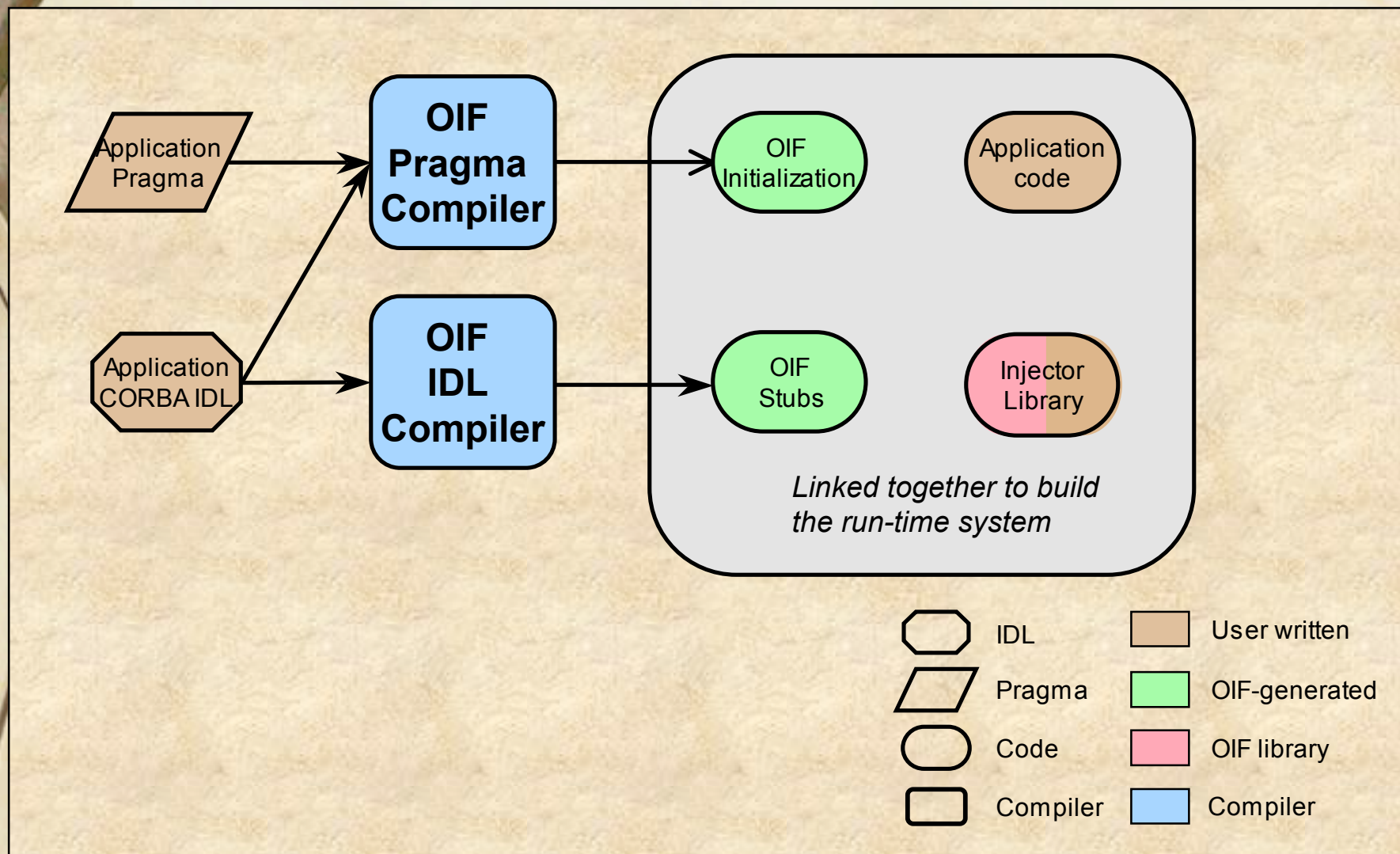
For each ility, we can declare a mapping from a location (on *method* in *class*) to how that ility is to be achieved.

Here we define the mapping from the achieve names to injector factories. The "do" clauses specify a partial ordering on the injectors.

```
policy vendoom is
  import vendoom;
  import injectors;
  ility Context, Security, QualityOfService,
    Reliability, Efficiency;
  var priority : int = {1};
  var retries : int = {5} only from client;
  for Context do copyContext;
  for Security on request in Controller do iButton;
  for QualityOfService on call in ByPriorityController
    do queuing;
  group cached Stuff on identifier, on description, on
    valueTo;
  for Efficiency within cachedStuff do caching;
  for Reliability do retry ;
  define copyContext for Context as
    client ContextInjectorFactory do first,
    server ContextInjectorFactory do last;
  define iButton for Security do last as
    client server injectors.
    AccessControlPkg.AccessControlInjectorFactory,
    client server injectors.
    IdentificationPkg.IButtonIdentificationInjectorFactor
    y;
  define queuing for QualityOfService as
    server injectors.
    QManager.QueueManagerInjectorFactory;
  define caching for Efficiency as
    client CacheInjectorFactory do after copyContext;
  define retry for Reliability as
    client ErrorRetryInjectorFactory ( retries = {"5"} )
    do last;
end;
```

Pragma Example

Compilation Process



Part 2: Aspect-Orientation





Separation of concerns

- A fundamental engineering principle is that of *separation of concerns*
 - Realizing different system concepts as separate, weakly linked elements
 - Distribution of expertise
- Separation of concerns promises better
 - Maintainability
 - Evolvability
 - Reusability
 - Adaptivity
- Concerns occur at both the
 - User/requirements level
 - Design/implementation level



Examples of Software Concerns


- Security
 - Always call the security check before allowing database access
- Accounting
 - Always debit the user's account on each access to a service of objects in the class...
- Synchronization
 - Don't let multiple users call any of methods f, g, or h on a single object simultaneously
 - The effects of these actions should be transactional
- Quality of service
 - Queue up the waiting calls handling them by priority
- Reliability
 - Provide replicants of this object
- Performance enhancements
 - Cache the results of calls to elements in this class
 - Display routines should show the results of changes, except display routines called in the scope of other display routines should buffer their changes for display all at once



Aspect-Oriented Programming

- Concerns crosscut
 - Apply to different modules in a variety of places
- Concerns must be composed to build running systems
- Aspect-Oriented Programming (AOP) is centered on
 - Separate expression of crosscutting concerns
 - Mechanisms to weave the separate expressions into a unified system
- Alternatives for specifying, modularizing and organizing software systems

In conventional programming, the code for different concerns often becomes mixed-together
(*tangled*)



Aspect-Oriented Software Development

- Aspect-Oriented Software Development is concerned with applying crosscutting separation of concerns technology throughout the software lifecycle
- More on this later



Real AOP Value

- We don't have to define all these policies before building the system
- Developers of tools, services, and repositories can remain (almost) completely ignorant of these issues
- We can change policies without reprogramming the system
- We can change policies of a running system



AOP Technology

- Software engineering technology for separately expressing systematic properties while nevertheless producing running systems that embody these properties
- Need to express
 - Base program
 - Separate concerns
 - How the separate concerns map to the base program
 - Or, if you prefer, just a jumble of program elements that must be combined.



Traditional Separation of Concerns

- Subprograms (procedures, functions, methods)
- Inheritance
- These do a good job of concern separation, but
 - The programmer has to explicitly invoke the desired behavior
 - The programmer has to always be aware of when to invoke what behavior
 - Changing a policy (that's not already embodied in a subprogram) requires finding all the places that need modification and changing them
- AOP is an alternative to this regime



AO Terminology 1

- Concern: Something we care about in an engineering process.
- Crosscutting concern: Concerns that in a given organization (modularization) of a system find themselves scattered throughout that organization.
 - What is crosscutting is a function of both the particular decomposition of a system and the underlying support environment.
- Code tangling: Ordinarily, implementing crosscutting concerns usually results in code tangling
 - Would prefer cohesive modules with simple interfaces
- Aspect: An aspect is a modular unit designed to implement a concern.



A0 Terminology 2

- **Join point:** A well-defined place in the structure or execution flow of a program where additional behavior can be attached.
- **Join point model:** The kinds of join points a system allows
 - ❑ Examples: method calls, field definition, access and modification; exceptions; and execution events and states.
- **Advice:** The behavior to execute at a join point.
 - ❑ Advice is often used to realize the behavior that satisfies a concern
 - ❑ Advice can run before, after, around, instead of or concurrently with the original behavior



A0 Terminology 3

- *Pointcut designator*: A description of a set of join points.
 - Usually, a universal quantification
- *Composition*: Bringing together and causing to interact separately defined software elements.
- *Weaving*: The process of composing core functionality modules with aspects, thereby yielding a working system.



A0 Terminology 4

- *Statics and Dynamics:* When decisions are made
 - Static elements are ones that can be determined before the program begins execution,
 - Typically at compile time
 - Dynamic elements happen at execution.
 - This can apply both to
 - Weaving
 - The join point model



A0 Terminology 5

● *The Tyranny of the Dominant Decomposition:*

- ❑ Base programs with aspects vs..
- ❑ Uniform element soup
- ❑ Key subtexts
 - Do aspects apply to aspects?
 - Is there a separation between describing what aspects exist and how they apply to the system?

The space of AOP language design

*In programs P , whenever condition C arises,
perform action A .*

- Dimensions of concern for the designer and implementer of an AOP system:
 - Quantification: What kinds of conditions C can be specified.
 - Interaction: What is the interface of the actions A . That is, how do they interact with base programs and each other.
 - Weaving: How will the system arrange to intermix the execution of the base actions of P with the actions A .



Quantification

- Join point model
- Over which elements and events can one quantify
 - Static quantification refers to elements recognizable in the source code
 - Lexical structure (bad idea)
 - Syntactic Structure
 - Compiler semantic structure
 - Dynamic quantification refers to the pattern of dynamic execution events



Interaction

- The structure of the aspect code
- Interactions among aspects
 - ▣ Including which runs first and how conflicts are recognized and resolved
 - ▣ Ordering
- How aspects communicate with each other and the base code
 - ▣ Visibility
- Aspect parameterizations



Weaving

- How does the system arrange to intermix the aspect and base behaviors
 - ❑ Compilers
 - ❑ Link-level wrapping
 - ❑ IDL compilers
 - ❑ Object-code modifiers
 - ❑ Meta-interpreters




AOP approaches

- Wrapping technologies
 - Composition filters
 - OIF
- Frameworks
 - Aspect-Moderator Framework
 - JAC
- Compilation technologies
 - AspectJ
 - HyperJ
- Post-processing strategies
 - JOIE
 - JMangler
- Event-based
 - EAOP
- Meta-level strategies
 - Heron
 - QSOUL/Logic Meta-Programming



Quantification and Obliviousness

- The unifying element of these approaches (and the characterizing definition of AOP) is the ability to state universally quantified programmatic assertions (quantification) on programs that have not been explicitly prepared to receive these assertions (obliviousness).
 - Quantification: A given assertion can have effect in many places in the system
 - Obliviousness: One can't tell for examining the local program source that the aspect will be invoked.
 - Surgery



Aspect-Oriented Software Development

- In general, programming is about realizing a set of requirements in an operational software system.
 - An evolving set of desired properties
 - Build a system to meet those requirements
- Software engineering is the accumulated set of processes, methodologies, and tools to ease that evolutionary process, including techniques for figuring out what it is that we want to build and mechanisms for yielding a higher-quality resulting system.
- Aspect-Oriented Software Development is concerned with applying cross-cutting-separation-of-concerns technology throughout the software lifecycle




Software Lifecycle

- Requirements
- Specifications
- Design
- Implementation
- Testing/validation
- Maintenance
- Evolution



Software Qualities

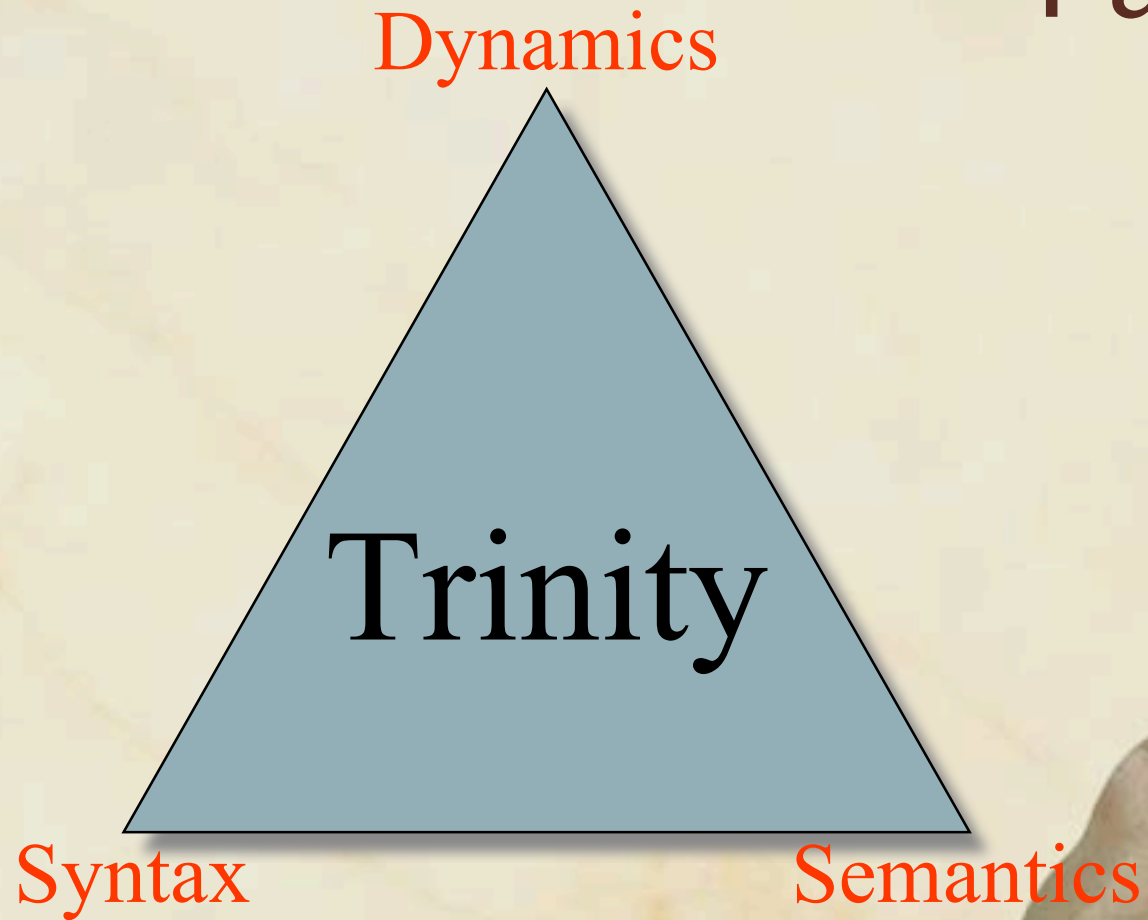
- correctness
- efficiency
- maintainability
- portability
- reliability
- predictability
- interoperability
- fault tolerance
- recoverability
- learnability
- analyzability
- adaptability
- reusability
- robustness
- testability
- verifiability
- comprehensibility
- consistency
- traceability
- evolvability
- measurability
- modularity



Aspect-Oriented Software Development

- Work on (almost) all the elements of the lifecycle
- Most work on
 - Process (often tied to CASE tools)
 - Modeling
 - UML modifications and uses for AO
- Starting to see work on
 - Testing
 - Evolution


Part 3





Over What can one Quantify?

- What is the space of possible join point models?
 - Static structure of the program
 - Lexical structure (bad idea)
 - Syntactic Structure
 - Static (semantic) compiler analysis
 - Things that happen during the execution of a system



Things that Happen during Program Execution: Events

- Events are with respect to the abstract interpreter of a language
- Software dark matter
 - Garbage collection
 - Thread swapping
- Unfortunately, language definitions don't define their abstract interpreters.



What can one change?

- Structural changes
 - ▣ Insert, delete, embed, rename
- Behavioral changes
 - ▣ Before, after, embed, elide




Trinity — An Extreme Experiment

- The extreme of expressiveness in quantification is to be able to quantify over everything statically deducible from the program and over all the history of events in a program execution
- Map from natural expressions to changes in the underlying system



Research regime

- Define a language of events and actions on those events.
- Determine how each event is reflected (or can be made visible) in source code.
- Create a system to transform programs with respect to these events and actions.
- Developing an environment for experimenting with AOP languages (DSL for AOP)




The Elements of Discourse

- *Syntactic*: Elements recognizable in the parse structure of a program
 - While loops
 - The parameter “x”
- *Semantic*: Elements recognizable by doing static semantic analysis
 - All references to this field or type
- *Dynamic*: events arising out of program execution.
 - Objects satisfying a property
 - Calls within a particular stack context
 - Dark matter
 - History



Actions

- Changes are realized by
 - Transforming the program
 - Inserting/calling additional behavior
 - Libraries



Goal

- Want to be able to uniformly describe desired changes about a system and have these changes realized by
 - Transforming the original code
 - Including inserting run-time checks for dynamic conditions
- Want to be able to “smoothly” express anything a programmer might want to say about a program
 - Part of the complexity is integrating statements over different kinds of things



Kinds of Events

- Events that correspond to the execution of a particular instruction
 - “The setting of X”
- Events that correspond to the transition of a predicate from false to true
 - When the value of $X + Y$ gets to be greater than 10.
- Events restricted to being achieved within a particular class or thread
- Events over multiple other events
 - Predicates on the entire execution history
- Dark matter events



Shadows

- The shadow of a quantification is the places in the code that might need to be changed to realize the quantification
- Sometimes there is a strong correspondence between syntactic structures, semantic objects and dynamic events
 - Sometimes there's not
- Sometimes one can shrink the shadow by more detailed analysis
 - Parallel to array bounds analysis in compilers
- For some predicates, the shadow may be (unrealistically) large
 - All potential null pointer exceptions



Elements and Shadows

- The shadow of syntactic predicate is usually just the syntactic element that satisfy that predicate
- The shadow of a semantic predicate is often the corresponding syntactic elements, but sometimes these elements don't exist in the text of the program
 - An inherited method
 - The implicit call to the super constructor at the beginning of a constructor



Events and shadows

- Dynamic shadows can be complex, and may include
 - Single concepts that are spread out over distributed loci
 - Tempering the semantics to make assumptions about
 - Unexaminable code
 - Aliasing
 - Preserving state
 - Temporal logic
 - Code reorganization and inferring properties for dark matter analysis



Trinity behavior

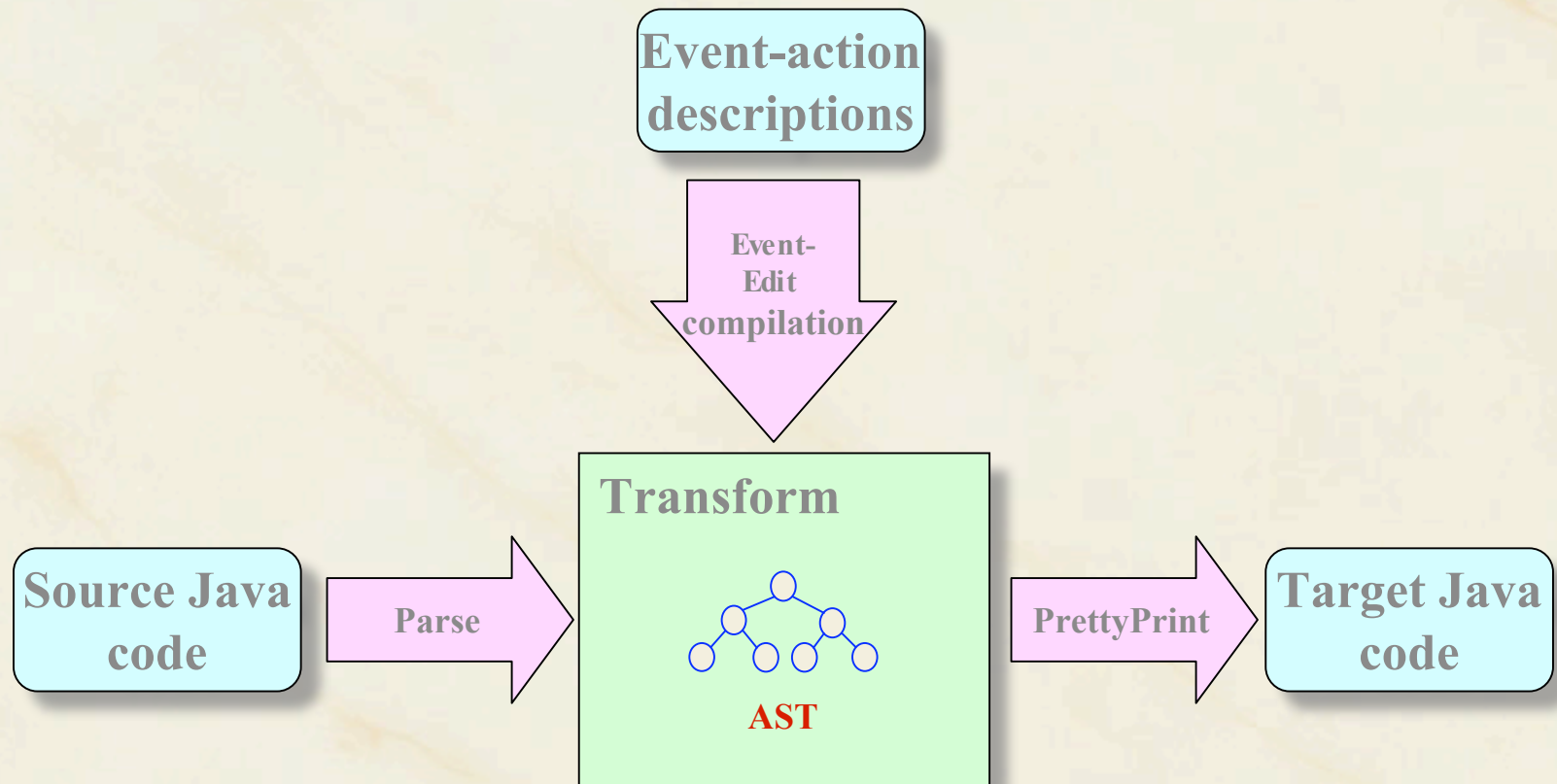
- Transform programs based on pattern-action rules
 - When the pattern of a quantification is seen, transform the program to perform the behavior desired in the action
 - Rules like database queries
 - Predicates include
 - Logical connectives
 - Temporal connectives
 - Program events including expression transition
- Transformations can be either
 - Structural: change the original program
 - Behavioral: perform some action before, after, around or instead of an original target
- Structural changes on events don't make sense



Transformational Alternatives

- For Java, can transform at
 - The source-code level
 - Human understandable
 - Can express some things that can be lost by the compilation process
 - Can work on code that won't compile
 - Output usable in all source-level tools
 - The byte-code level
 - More reliably captures certain events
 - Works even when lacking source code

Architectural View





Applications

- Debugging
- Profiling
- Monitoring
- Contextual evaluation (the "jumping beans" problem)
- Autonomic computing
- Security
- Concurrency
- Resource management
- Refactoring
- Persistence
- User interface consistency

Closing Remarks



Readings



ASPECT-ORIENTED Software Development

ROBERT E. FILMAN
TZILLA ELRAD
SIOBHÁN CLARKE
MEHMET AKŞIT



Readings

- AOP in general
 - Robert E. Filman, Tzilla Elrad, Siohbán Clarke, and Mehmet Aksit (Eds.) Aspect-Oriented Software Development, Addison-Wesley, 2005
 - Communications of the ACM, Oct 2001 theme issue on AOP
- AOP Is
 - Robert E. Filman and Daniel P. Friedman. "Aspect-Oriented Programming is Quantification and Obliviousness." Chapter 2 in AOSD book
- OIF
 - Robert E. Filman, Stu Barrett, Diana D. Lee, and Ted Linden. Inserting Ilities by Controlling Communications. Communications of the ACM, Vol. 45, No. 1, January, 2002, pp. 116–122.
 - <http://ic.arc.nasa.gov/people/filman/text/oif/cacm-oif.pdf>
- Event-based AOP
 - Robert E. Filman and Klaus Havelund. Source-Code Instrumentation and Quantification of Events. AOSD 2002 Workshop on Foundations Of Aspect-Oriented Languages (FOAL), Twente, Netherlands, April 2002.
 - <http://ic.arc.nasa.gov/~filman/text/oif/aop-events.pdf>



More readings: AOP Bibliography

- www.aosd.net/technology/aosd-bibliography.pdf
- www.aosd.net/technology/aosd-bibliography.bib

Questions/Discussion

